Fast Reachability Query Processing

Jiefeng Cheng, Jeffrey Xu Yu, and Nan Tang

The Chinese University of Hong Kong, China
{jfcheng, yu, ntang}@se.cuhk.edu.hk

Abstract. Graph has great expressive power to describe the complex relationships among data objects, and there are large graph datasets available. In this paper, we focus ourselves on processing a primitive graph query. We call it *reachability query*. The reachability query, denoted $A \rightsquigarrow D$, is to find all elements of a type D that are reachable from some elements in another type A. The problem is challenging because the existing structural join algorithms, studied in XML query processing, cannot be directly applied to it, because those techniques make use of the tree-structure heavily. We propose a novel approach which can process reachability queries on the fly while keeping the space consumption small that is needed to keep the required information for processing reachability queries. In brief, our approach is based on 2-hop labeling for a directed graph G which consumes $O(|V| \cdot \log |E|)$ space. We construct a novel join-index which is built on a small table and B+-tree. With the join-index, the high efficiency is achieved. We conducted extensive experimental studies, and we confirm that our approach can efficiently process reachability queries over a graph or a tree.

1 Introduction

Due to the advanced Web technology and the new techniques for data archiving and analyzing, there is a huge volume of data available in public, which are graph structured in nature including hypertext data, semi-structured data and XML [1]. For instance, in Web mining, the navigation patterns of Web surfers is modeled as directed acyclic graphs to improve the analysis of the navigation behavior of user groups [4]. In Genome biology, graph and network models have been used, for example, gene-regulatory networks or metabolic networks. HumanCyc [13] and Cyc [11] are two such databases where graph is used to represent their inter-reactions.

Graph has great expressive power to describe the complex relationships among data objects. Real applications use different facilities/systems to handle their data as either directed graphs, or directed acyclic graphs, or trees. As a major standard for representing data on the World-Wide-Web, XML provides facilities for users to model data as trees or to view data as graphs with two different links, the parent-child links (document-internal links) and reference links (cross-document links) where the cross-document links are supported by value matching using ID/IDREF in XML. In addition, the XLink (XML Linking Language) [8] and XPointer (XML Pointer Language) [9] provide more opportunities for users to manage their complex data as graphs and integrate data effectively.

In this paper, we study a primitive graph query, called *reachability query*, that is needed in any types of graphs (directed graphs, directed acyclic graphs, or trees). In

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 674–688, 2006.

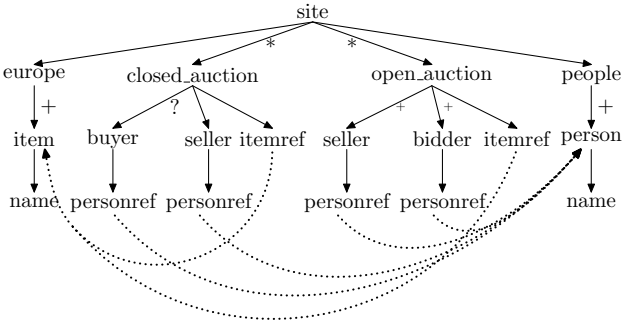


Fig. 1. A part of DTD used in XMark Benchmark [16]

brief, given two types of elements in a graph, A and D, the reachability query, denoted $A \rightsquigarrow D$, is to find all elements of the type D that are reachable from some elements in the type A.

The needs of such a reachability query can be even found in XML. Consider the DTD graph in Fig. 1, which is a part of DTD used in XMark Benchmark [16]. The DTD specifies `person`, `seller`, and `bidder`, where a `person` has a `name`, and a `seller`/`bidder` may have ID/IDREF to reference a `person`, because `seller` and `bidder` are `person`. In Fig. 1, solid links represent document-internal links whereas dashed links represent cross-document links (ID/IDREF). Suppose that we want to know seller's names. It becomes difficult to find `names` of `sellers` using the XPath operator // (descendants-or-self-axis) alone, because // does not traverse the cross-document links supported by ID/IDREF. It needs more joins to process this reachability query using XPath even the underneath DTD is known. It requests even great effort to process a reachability query if there is no DTD available, and may need to traverse the whole graph. It is unreasonable for users not to be able to find such information in a graph effectively and efficiently, where users are recommended to model data by sharing the commonly-used parts (e.g., seller as person) as much as possible.

It is important to note that the reachability query we are studying is a more general facility to find relevant information in arbitrary graphs (including trees). $A//D$ is a special case of $A \rightsquigarrow D$ when data is a tree. There exist many advanced techniques to process structural join or containment join, $A//D$, for XML including [3, 6, 10]. Those techniques cannot be directly applied to reachability queries, $A \rightsquigarrow D$, because they all at maximum make use of the XML tree-structure without ID/IDREF. As an exemption, in [17], Wang et al studied reachability queries, $A \rightsquigarrow D$, and proposed two algorithms based on a graph coding scheme [2] and structural join techniques used in XML.

In this paper, we propose a novel index-based approach that can efficiently process reachability queries, $A \rightsquigarrow D$, over a graph (including tree). Our approach achieves high efficiency while keeping space consumption small. First, we adapt a graph coding scheme which consumes $O(|V| \cdot \log |E|)$ space for a directed graph $G(V, E)$. The coding scheme is based on the 2-hop labeling for a graph [7]. We report a simple yet effective technique that can further reduce the space consumption of 2-hop cover. Note: There are several recent works on computing 2-hop cover using a divide-and-conquer approach [14, 15] or a geometry-based approach [5]. The incremental maintenance of

such 2-hop cover was also reported in [15]. Second, we propose an algorithm for reachability queries based on 2-hop labeling. In order to be more efficiently processing reachability queries, we construct a novel join index consisting of a table and a B+-tree. Our join index allows us to process any $A \rightsquigarrow D$ queries in a graph on the fly. As shown in our extensive experimental studies, our index-based approach consumes the similar amount of space like the interval-based graph coding scheme as reported in [17], and significantly outperforms the algorithms given in [17]. Third, we confirm that our approach can also process structural joins for XML tree data efficiently.

The rest of the paper is organized as follows. Section 2 discusses graph data and defines reachability queries. Section 3 introduces the only existing algorithm for reachability queries over graphs, which serves the baseline for us to compare. Section 4 discusses our approach in detail. We will give the graph coding we used, the query processing techniques and its join index. Experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

2 Graph Data and Reachability Query

We consider a large directed node-labeled graph $G = (V, E, \lambda, L)$, to represent a complex data set which can be semi-structured data or XML data. Here, V is a set of elements, E is a set of edges, L is a set of labels, and λ is a function which assigns a node a label. Given a label $l \in L$, the extent of l is defined as a set of nodes whose label is l, denoted $ext(l)$. An example is given below.

Example 1. *A simple XML document is shown in Fig. 2. The example shows two relationships between elements in XML. One represents the parent-child relationship between two elements in XML. For example, the element* `<site>` *has* `<europe>` *as its subelement (parent-child). The parent-child relationships for an XML document form a tree representation. Fig. 3 (a) shows the tree representation of Fig. 2. The other is a general reference from one element to another using* `id` *and* `idref` *based on value-matching, where* `id` *specifies a unique user-assigned identifier for an element, and* `idref` *specifies a link to the element with the same identifier. For example, in Fig. 2, line-4, an item is associated with an identifier (*`id="item1"`*). In the middle of Fig. 2, there is an element* `itemref` *which points to (*`id="item1"`*) using* `idref="item1"`*. Both parent-child and id-idref relationships together form a graph representation for an XML document. Fig. 3 (b) shows the graph representation of Fig. 2. Note: in both Fig. 3 (a) and (b), every node is associated with a system-assigned object identifier, for example 16.*

Given a directed node-labeled graph $G = (V, E, \lambda, L)$, we define a *reachability query*, denoted $A \rightsquigarrow D$, as to find whether elements with label A can reach elements with label D. In other words, let A and D be two labels in L. $A \rightsquigarrow D$ finds all pairs of (u, v) such as v is reachable from u where $u \in ext(A)$ and $v \in ext(D)$ in the graph G.

We consider two cases for a query $seller \rightsquigarrow name$ against the simple XML document (Fig. 2). First, we issue the query against its tree representation (Fig. 3 (a)), the query result is empty, because there does not exist a data path from an element of `seller` to an element of `name`. It is important to note that, when XML is modeled as

```
<site>
  ... ...
  <Europe>
    <item id="item1">
      <name>desktop</name>
    </item>
    <item id="item2">
      <name>laptop</name>
    </item>
  </Europe>
  <closed_auction>
    <buyer>
      <personref idref="person1"/>
    </buyer>
    <seller>
      <personref idref="person2"/>
    </seller>
    <itemref idref="item1"/>
  </closed_auction>
  <open_auction>
    <seller>
      <personref idref="person1"/>
    </seller>
    <bidder>
      <personref idref="person2"/>
    </bidder>
    <itemref idref="item2"/>
  </open_auction>
  <people>
    <person id="person1">
      <name>Joe</name>
    </person>
    <person id="person2">
      <name>Ray</name>
    </person>
  </people>
</site>
```

Fig. 2. A Sample XML Document Fragment

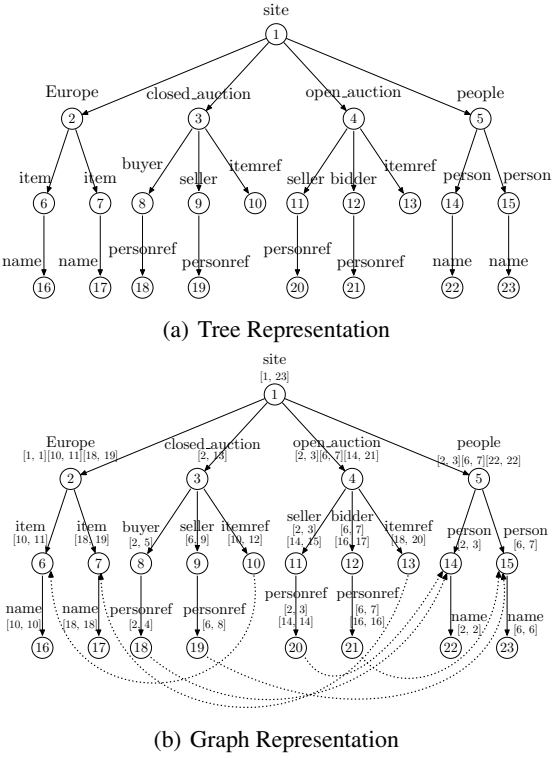


Fig. 3. Document Representation

a tree, $A \rightsquigarrow D$ is the same as $A//D$ used in XPath queries. Second, we issue the query against its graph representation (Fig. 3 (b)), there are two pairs, namely, (9, 23) and (11, 22), because sellers are persons and therefore persons names are sellers names. An acute reader may ask whether the second case can be processed using // in XPath with value index efficiently. The answer is rather negative. For getting the sellers names, 22 and 23, the XQuery is as follows.

```
for $p in //seller
let $n := for $s in //person
          where $p/personref/@idref = $s/@id
return $n/name
```

The above XQuery finds all `sellers` and all `persons` followed by id/idref matching, and is time-consuming. It is worth of noting that it requests users to know the underneath schema of XML data. In fact, even when a user knows the underneath schema, sometime, it is rather complicated to process $A \rightsquigarrow D$ using // and id/idref. For example, the reachability query, $closed\_auction \rightsquigarrow name$, requires more joins on id/idref if it is processed using XQuery.

3 An Existing Approach

For processing reachability queries, $A \rightsquigarrow D$, two algorithms were processed in [17] using a structural join method based on an interval-based coding scheme for arbitrary directed graphs. In brief, given a directed graph, G. First, it constructs a directed acyclic graph G' by condensing a maximal strongly connected component in G as a node in G'. Second, it generates interval-based codes for G' the directed acyclic graph based on [2]. Third, all nodes in a strongly connected component in G share the same code assigned to the corresponding representative node condensed in G'.

Agrawal et al [2] proposed a method for labeling directed acyclic graphs using intervals. The labeling is done in three steps for a directed acyclic graph, G_D. 1) Construct an optimum tree-cover $\mathcal{T}$. An optimum tree-cover is defined as to minimize the number of intervals. 2) Every node, v, in the $\mathcal{T}$ is labeled using an interval $[s, e]$. A node v has a *postorder number*, denoted *po*, which is the number assigned following a postorder traversal of the tree starting from 1. The e value in $[s, e]$ for a node v is the postorder number of the node v, and the s value in the interval is the smallest postorder number of its descendants, where $s = e$ if v is a leaf node. 3) After $\mathcal{T}$ is labeled, it examines all nodes of $G_D$in the reverse topological order. During the traversal, for each node u, add all the intervals associated with v, if there exists an edge (u, v), into the interval associated with u. Note: an interval can be eliminated if it is contained in another. Let I_u be a list of intervals assigned to a node u. Suppose there are two nodes u and v where $I_u = \{[s_1, e_1], [s_2, e_2], \cdots, [s_n, e_n]\}$, and $I_v = \{[s'_1, e'_1], [s'_2, e'_2], \cdots, [s'_m, e'_m]\}$. There exists a path from u to v if the postorder of v is in an interval, $[s_j, e_j]$, of u. The interval-based graph code for the sample XML documents as a directed graph is given in Fig. 3 (b).

Two algorithms, *GMJ* and *IGMJ*, were studied in [17]. We introduce the algorithm *IGMJ* (Improved Graph-Merge Join) below, because it outperforms *GMJ*.

As shown in Algorithm 1, for $A \rightsquigarrow D$, *IGMJ* takes two lists $Alist$ and $Dlist$ as its input. Here, $Alist$ and $Dlist$ are two lists of nodes belonging to label A and D respectively. $Alist$ is sorted on the intervals $[x, y]$ by the ascending order of x and then

Algorithm 1. *IGMJ* $(Alist, Dlist)$

```
1: a := Alist.head();   d := Dlist.head();
2: R := ∅;
3: while a ≠ ∅ ∨ d ≠ ∅ do
4:    if a.x ≤ d.postid then
5:       rstree.trim(a.x);   rstree.insert(a);   a := a.next();
6:    else
7:       rstree.trim(d.postid);
8:       for all elements a ∈ rstree do
9:          R.insert(a.id, d.id);
10:      end for
11:      d := d.next();
12:   end if
13: end while
14: return R;
```

the descending order of y. If a node of A has n intervals, then it will have n entries in the $Alist$. $Dlist$ is sorted by the ascending order of postorder. The motivation is to leverage the order in the intervals and postorder to accelerate join processing. The $rstree$ is a range search tree. In the range search tree, the intervals are indexed and organized according to their y values. Two operations, $trim(v)$ and $insert(n)$, are used such as $trim(v)$ is to batch delete the intervals whose y values smaller than v and $insert(n)$ is to insert a node to the range search tree.

4 A Novel Join Index

In this paper, we propose a novel join index for a set of reachability queries $A \rightsquigarrow D$ in a graph $G = (V, E, \lambda, L)$. Note, there are in total ${}_nC_2$ combinations of such $A \rightsquigarrow D$ if n is the size of the set of labels, L. The goal is to minimize the query processing cost for any reachability query $A \rightsquigarrow D$ in a graph G on the condition that the size of the join index is small.

Intuitively, it is seen as difficult to minimize both query processing cost and the space required for such a join-index. Let's consider the simplest approach. That is to materialize all ${}_nC_2$ combinations of such $A \rightsquigarrow D$ on disk, $\langle L_1, L_1\rangle, \langle L_1, L_2\rangle, \cdots, \langle L_{n-1}, L_n\rangle, \langle L_n, L_n\rangle$ if there are n labels. In doing so, we can answer any reachability query, $A \rightsquigarrow D$, by simply loading the corresponding materialized join-index from disk. However, the simplest approach here processes reachability queries efficiently at the expense of huge disk space. Suppose that there are n labels, The worst case is that the extent of a label, $ext(L_i)$, may occur n times because there are n reachability queries from L_i to any other L_j for $j = 1, \cdots, n$. It also implies that the join-index can be possibly n times of $|V|$ where V is the set of nodes in graph G. Such simplest approach is infeasible.

In the following, we propose a novel join index which takes space less than that of the data in general. We will show that we can process any reachability query efficiently with the proposed join index. Below, we will first introduce a graph coding, our query processing techniques based on the graph coding, and the join index structure.

4.1 2-Hop Reachability Labeling

The 2-hop reachability labeling is defined in [7]. We introduce it below in brief. Let $G = (V, E)$ be a directed graph. A 2-hop reachability labeling on graph G assigns every node $v \in V$ a label $L(v) = (L_{in}(v), L_{out}(v))$, where $L_{in}(v), L_{out}(v) \subseteq V$, such that every node x in $L_{in}(v)$ connects to every node y in $L_{out}(v)$. A node v is reachable from a node u, denoted $u \rightsquigarrow v$, if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The size of the 2-hop reachability labeling over a graph $G(V, E)$, is given as L, below.

$$L = \sum_{v \in V(G)} |L_{in}(v)| + |L_{out}(v)| \tag{1}$$

Consider $L(v)$ for a node v, the name of 2-hop comes from the idea that the reachability from a node $x \in L_{in}(v)$ to node $y \in L_{out}(v)$ is via the node v in the middle way, so

that the first hop is from x to v and the second hop is from v to y. In order to solve the 2-hop reachability labeling, Cohen et al introduce 2-hop cover which is a set of paths in the graph G. The definition is given below [7].

Definition 1 (2-hop cover). *Given a directed graph $G = (V, E)$. Let $P_{u \rightsquigarrow v}$ be a set of paths from node u to node v in G, and P be a set of all such $P_{u \rightsquigarrow v}$ in G. A hop, h_u, is defined as $h_u = (p_u, u)$, where p_u is a path in G and u is one of the endpoints of p_u. A 2-hop cover, denoted H, is a set of hops that covers P, such as, if node v is reachable from node u then there exists a path p in the non-empty $P_{u \rightsquigarrow v}$ where the path p is concatenation of p_u and p_v, denoted $p = p_u p_v$, and $h_u = (p_u, u)$ and $h_v = (p_v, v)$.*

The 2-hop reachability labeling can be derived from a 2-hop cover [7]. Formally, given a 2-hop cover H, the label for node v, $L(v) = (L_{in}(v), L_{out}(v))$ becomes $L_{in}(v) = \{x \mid ((x, v), v) \in H\}$ and $L_{out}(v) = \{x \mid ((v, x), v) \in H\}$. Given a 2-hop reachability labeling, $H = \cup_{v \in V} H_v$ where $H_v = \{((x, v), v) | x \in L_{in}(v)\} \cup \{((v, x), v) | x \in L_{out}(v)\}$. In addition, the size of the 2-hop cover, $|H|$, for a graph G, is the same as that of 2-hop reachability labeling ($|H| = L$).

The 2-hop cover problem is to find the minimum size of 2-hop cover for a given graph $G(V, E)$, which is proved to be NP-hard [7]. Cohen et al show that a greedy algorithm exists to compute a near optimal solution for the 2-hop cover problem. The resulting size of the greedy algorithm is larger than the optimal at most $O(\log n)$ where $n = |V|$. In [14, 15], Schenkel et al studied a partition-based approach to efficiently generate 2-hop cover for a large graph. The incremental maintenance of such 2-hop code for a graph was also discussed in [15]. In [5], we also propose a novel geometry-based algorithm to improve the efficiency of computing 2-hop cover for even large dense graphs.

The computed 2-hop labeling for the graph representation of the XML document (Fig. 3 (b)) is listed in Table 1. Here, The table has four attributes L (label), v (node), L_{in} of v and L_{out} of v. For example, $8 \rightsquigarrow 14$, because $L_{out}(8) \cap L_{in}(14) = \{8\} \neq \emptyset$.

We observe that the 2-hop labeling can be further compressed by removing hops in the form of $(v \rightsquigarrow v, v)$. Table 2 shows the compact 2-hop labeling for the sample example. We use the compact 2-hop labeling in our work. When compact 2-hop labeling is used, the condition of checking reachability needs to be modified as below: if $u \rightsquigarrow v$ is true, then one of the three conditions must be true, (a) $L_{out}(u) \cap L_{in}(v) \neq \emptyset$, (b) $u \in L_{in}(v)$ and (c) $v \in L_{out}(u)$. As can be seen in Table 2, $8 \rightsquigarrow 14$ is true, because $8 \in L_{in}(14)$.

Table 1. 2-hop labeling for the graph in Fig. 3 (b)

$\mathcal{T}$	v	L_{in}	L_{out}	$\mathcal{T}$	v	L_{in}	L_{out}	$\mathcal{T}$	v	L_{in}	L_{out}
site	1	$\emptyset$	2, 3, 4, 5	item	7	2, 4	$\emptyset$	person	15	3, 4, 15	15
europe	2	2	2	buyer	8	3, 8	8	name	16	2, 3, 6	$\emptyset$
closed_auction	3	3	3	seller	9	3	15	name	17	2, 4, 7	$\emptyset$
open_auction	4	4	4	seller	11	4	14	name	22	3, 4, 8, 14	$\emptyset$
people	5	5	5, 14, 15	bidder	12	4	15	name	23	3, 4, 15	$\emptyset$
item	6	2, 3, 6	6	person	14	3, 4, 8, 14	15				

Table 2. Compact 2-hop labeling for the graph in Fig. 3 (b)

L	v	L_{in}	L_{out}	L	v	L_{in}	L_{out}	L	v	L_{in}	L_{out}
site	1	$\emptyset$	$2,3,4,5$	item	7	$2,4$	$\emptyset$	person	15	$3,4$	$\emptyset$
europe	2	$\emptyset$	$\emptyset$	buyer	8	3	$\emptyset$	name	16	$2,3,6$	$\emptyset$
closed_auction	3	$\emptyset$	$\emptyset$	seller	9	3	15	name	17	$2,4,7$	$\emptyset$
open_auction	4	$\emptyset$	$\emptyset$	seller	11	4	14	name	22	$3,4,8,14$	$\emptyset$
people	5	$\emptyset$	$14,15$	bidder	12	4	15	name	23	$3,4,15$	$\emptyset$
item	6	$2,3$	$\emptyset$	person	14	$3,4,8$	$\emptyset$				

In the following, for simplicity, we use the non-compact 2-hop labeling to explain our ideas and we use the compact 2-hop labeling in our testing.

4.2 Reachability Query Processing Based on 2-Hop Labeling

We discuss reachability query processing below based on hops (Definition 1). Recall: given a hop, $h_u = (p_u, u)$, p_u is a path and u is one of the endpoints of p_u. Suppose w is the other endpoint. There are only two possible cases for p_u. One is $p_u = w \rightsquigarrow u$, and the other is $p_u = u \rightsquigarrow w$. Here, w is the center of the hop. In Table 1, all centers appear in either L_{in} or L_{out}. Given a set of hops, $\mathcal{H}$, constructed for a graph G. We can obtain such a set of centers, denoted W. For each $w \in W$, we can also construct two sets, $H^+(w)$ and $H^-(w)$ as below.

$$H^+(w) = \{u \mid w \rightsquigarrow u, (w \rightsquigarrow u, u) \in H\} \tag{2}$$

$$H^-(w) = \{u \mid u \rightsquigarrow w, (u \rightsquigarrow w, u) \in H\} \tag{3}$$

Here, $H^+(w)$ consists of all nodes that can be reached from the center w, and $H^-(w)$ consists of all nodes that can reach the center w. Both $H^-(w)$ and $H^+(w)$ together serve as an inverted index showing that every node in $H^+(w)$ can be reached from all nodes in $H^-(w)$. Furthermore, we can efficiently find the nodes in either $H^+(w)$ or $H^-(w)$ that are associated with a label L_i.

$$H^+(L_i, w) = \{u \mid u \in H^+(w) \wedge \lambda(u) = L_i\} \tag{4}$$

$$H^-(L_i, w) = \{u \mid u \in H^-(w) \wedge \lambda(u) = L_i\} \tag{5}$$

As an example, consider Table 1. $H^+(14) = \{22\}$ and $H^-(14) = \{5, 11\}$. It suggests that 22 can be reached from 5 and 11. Furthermore, we have $H^+(name, 14) = \{22\}$, $H^-(people, 14) = \{5\}$, and $H^-(seller, 14) = \{11\}$. It suggests that the `seller` identified by node identifier 11 has a `name` identified by 22.

Reachability query processing. Given the above equations, a reachability query, $A \rightsquigarrow D$, can be processed in two steps. First, it finds all centers $w \in W$ in which $H^+(A, w)$ and $H^-(D, w)$ are non-empty. Note: both A and D are labels. Second, it produces the query result by simply pairing every element in $H^-(A, w)$ with every element in $H^+(D, w)$, because they must be reachable. The correctness of our reachability query processing is ensured based on the correctness of 2-hop labeling. Recall: 2-hop labeling

Table 3. Code Sizes for different XMark Datasets

Name	Factor	Data Set (M)	Graph		Tree	
			2-Hop (M)	Interval (M)	2-Hop (M)	Interval (M)
10M	0.1	11.88	3.97	3.43	4.15	3.11
20M	0.2	23.93	7.97	6.89	8.35	6.26
30M	0.3	39.24	11.89	12.43	12.43	9.32
40M	0.4	47.51	15.83	13.66	16.53	12.40
50M	0.5	59.03	19.81	17.07	20.61	15.45

ensures that, if node v can be reachable from node u in a graph G, there must exist a center w such as w is reachable from u and v is reachable from w. We maintain the 2-hop labeling. For a reachability query $A \rightsquigarrow D$, we check all possible centers w, and we only report those elements of A that can reach elements D. In other words, we do not include any results which shall not be include and we do not miss any results.

The size of graph code. The code size of 2-hop labeling in general is similar with the interval-based code. We will discuss it in our experimental studies. As shown in Table 3, for example, when the XMark dataset [16] is 59.03MB, the size of 2-hop labeling is less than a half of the raw dataset, 19.81MB, whereas the interval-based code is 17.07MB. It implicitly suggests that the space needed for 2-hop labeling is reasonable small. In the next subsection, we discuss the efficiency of reachability query processing using an join index.

4.3 Join Index

As discussed in the previous section, our reachability query processing is done in two steps for $A \rightsquigarrow D$: i) finding all centers $w \in W$ in which $H^+(A, w)$ and $H^-(D, w)$ are non-empty, and ii) pairing every element in $H^+(A, w)$ with every element in $H^-(D, w)$. In order to process these two steps efficiently, we propose an join index which consists of two parts, namely, a center-table and a balanced B+-tree. The center-table is an implementation of a function, $W(A, D)$, which returns all centers $w \in W$ such as both $H^-(A, w)$ and $H^+(D, w)$ are non-empty. The B+-tree is illustrated in Fig. 4. The search key on the B+-tree is a pair $\langle L_i, w \rangle$, where L_i is a

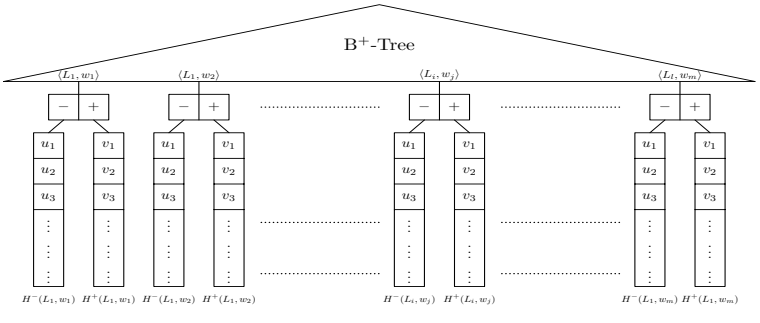


Fig. 4. B+-tree

Algorithm 2. *HPSJ*(A,D)

1: $\mathcal{W} \leftarrow W(A, D)$;
2: **for each** $w \in \mathcal{W}$ **do**
3: Search for $H^-(A, w)$ and $H^+(D, w)$ on the B$^+$-Tree;
4: pair every element in $H^-(A, w)$ with all elements in $H^+(D, w)$ and output the result;
5: **end for**

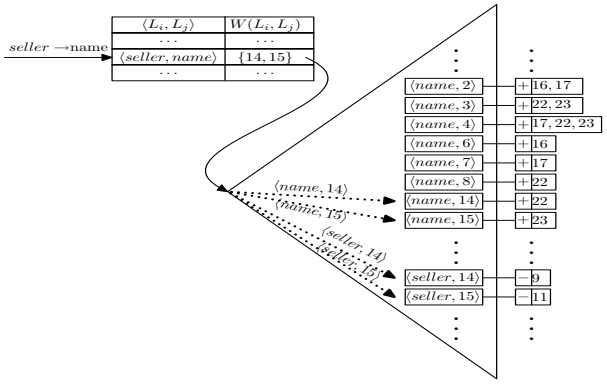


Fig. 5. An Example

label and w is a center. In the leaf nodes of B+-tree, there are two pointers for a search key $\langle L_i, w\rangle$, namely, $H^-_{L_i}(w)$ and $H^+_{L_i}(w)$. The B+-tree can be fast constructed using 2-hops, and is easy to maintain. Due to limit of space, we do not discuss the maintenance issues in this paper.

Our *HPSJ* algorithm is outlined in Algorithm 2, which takes two labels, A and D, as input, for a reachability query, $A \rightsquigarrow D$, and process $A \rightsquigarrow D$ over a graph which is stored in our join-index. In the algorithm, we first find the list of centers, $\mathcal{W}$, from the center-table (line-1). Note: $W(A, D)$ returns centers, w, where both $H^-(A, w)$ and $H^+(D, w)$ are non-empty. For each of the centers, $w \in \mathcal{W}$, we search the B+-tree part of the join-index; and report the join results by pairing every elements in $H^-(A, w)$ and $H^+(D, w)$.

As an example, we show how to process a reachability query, $seller \rightsquigarrow name$, against the graph representation of XML (Fig. 3 (b)), in Figure 5. First, we identify all the centers by calling $W(seller, name)$ using the center-table. We obtain two centers, 14 and 15 (Note: $\mathcal{W} = \{14, 15\}$). We then obtain all `seller` that can reach `name` in both centers, 14 and 15, by searching B+-tree. For example, with the search key $\langle saller, 14\rangle$, we find 9 in $H^-(seller, 14)$; with the search key $\langle name, 14\rangle$, we find 22 in $H^+(name, 14)$; and therefore, the seller (9) has a name (22).

5 Performance Study

Our approach can process reachability queries, $A \rightsquigarrow D$, over graphs and trees. We conducted extensive performance studies, in comparison with those algorithms that are

designed for reachability queries over graphs and reachability queries over trees, respectively. For the former, we compare our proposed *HPSJ* algorithm against the *IGMJ* algorithm [17] over large graphs. For the latter, we compare our *HPSJ* algorithm with two index-based structural join algorithms, *XR-tree* [10] and *B+-tree* [6] over trees. We report our results in this section.

The datasets we used for testing are generated using the XMark benchmark [16]. Five factors are used, namely, from 0.1 to 0.5. Given an XMark dataset generated with a factor. We generate graph data by treating both document-internal links (parent-child) and cross-document links (ID/IDREF) as edges in the same manner, and we generate tree data with the document-internal only. The details are given in Table 3. In Table 3, the first column is the dataset name for easy reference, the second column is the factor used to generate a XMark dataset. The third column shows the size of the dataset used. The fourth and fifth columns show the size of the compact 2-hop labeling and the interval [2], for the graph, respectively. The last two columns show the size of the compact 2-hop labeling and the interval [2], for the tree, respectively. All sizes are of the unit of Megabyte. It is worth noting that, for both graph and tree, the size of the 2-hop labeling and the size of the interval coding are marginally different. Given a graph and a tree generated from the same XMark dataset with a certain factor. The size of the 2-hop labeling code for the graph is smaller than that for the tree, because 2-hop labeling can cover a rather large number of nodes in graph, in comparison with the case of tree. On the other hand, the size of the interval code for the graph is larger than that for the tree, as expected.

We implemented our *HPSJ* algorithm and the *IGMJ* algorithm using MS VC++. We used the code implemented by Jiang and Wang [10] for testing *XR-tree* [10] and *B+-tree* [6] over trees. We conducted our testing on a PC with Pentium 2GHz CPU, 1G main memory and an 80G SCSI disk. The OS is Windows 2000 Professional.

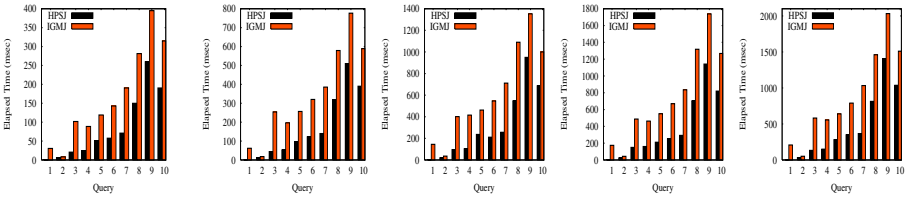
5.1 Reachability Query Processing over Graphs

For the XMark benchmark, there are many labels in its DTD. A part of the XMark DTD is given in Fig. 1. We conducted testing over a large number of combinations of two labels, and report 10 queries in this testing. The 10 queries are selected as they give different results with different distributions. Table 4 lists the 10 selected queries, in which the first column is the query id, the second column shows the reachability query, $A \rightsquigarrow D$, the following 5 pairs show the sizes of $|A|$ and $|D|$ for each XMark dataset, namely, 10M, 20M, etc, for graphs. Here, for $A \rightsquigarrow D$, $|A|$ indicates the total number of A elements that can match D elements, and $|D|$ indicates the total number of D elements that can match A elements.

We process the 10 selected reachability queries, using *HPSJ* and *IGMJ*. The performance results are shown in Fig. 6. There are 5 subfigures, for 5 different datasets, 10M, 20M, 30M, 40M and 50M, respectively, in Fig. 6. In each subfigure, x-axis is the 10 queries, and y-axis is the elapse time in msec. Our algorithm, *HPSJ*, significantly outperforms *IGMJ* in all cases, even though our algorithm uses a slightly larger size of graph labeling. We explain the reason below. Give a reachability query, $A \rightsquigarrow D$. There are $|W(A, D)|$ centers found in the first step in our algorithm. In the second step, we only need to search over the B+-Tree of our join-index for $|W(A, D)|$ times. We don't

Table 4. Queries for XMark Data Sets as Graphs

	$A \rightsquigarrow D$	10M		20M		30M		40M		50M	
		$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$
Q1	$africa \rightsquigarrow item$	1	55	1	110	1	181	1	220	1	269
Q2	$closed\_auctions \rightsquigarrow reserve$	1	592	1	1175	1	1891	1	2319	1	2854
Q3	$closed\_auctions \rightsquigarrow item$	1	2152	1	4317	1	7111	1	8626	1	10565
Q4	$europe \rightsquigarrow incategory$	1	2307	1	4490	1	7375	1	8974	1	10945
Q5	$namerica \rightsquigarrow incategory$	1	3756	1	7427	1	12531	1	15261	1	18630
Q6	$people \rightsquigarrow incategory$	1	4456	1	8888	1	14686	1	17997	1	21762
Q7	$closed\_auctions \rightsquigarrow bidder$	1	6059	1	11937	1	18889	1	23077	1	28755
Q8	$item \rightsquigarrow keyword$	2055	4331	4199	8627	6855	14233	8317	17128	10148	20844
Q9	$item \rightsquigarrow text$	2175	6462	4350	12830	7177	21257	8700	25664	10657	31371
Q10	$item \rightsquigarrow incategory$	2175	8219	4350	16290	7177	27037	8700	32787	10657	40232

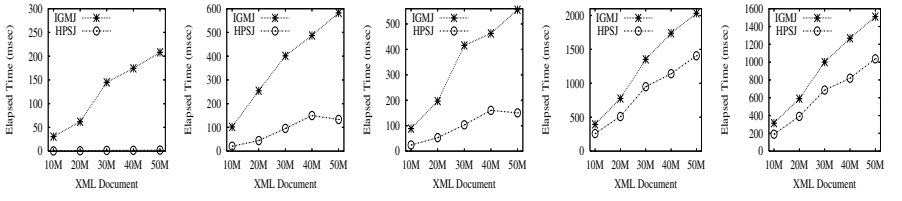


(a) 10M (b) 20M (c) 30M (d) 40M (e) 50M

Fig. 6. Reachability Query Processing over Different Graphs

need to check any join conditions but simply pair the results retrieved from B+-tree. Note: the size of $|W(A, D)|$ has the impacts on the number of searching over B+-tree used in our join-index. *IGMJ* behaves like a structural join algorithm, which has to scan the two interval lists, A and D, once. *IGMJ* needs to check join condition with some run-time data structures to support it as shown in Algorithm 1. The cost depends on the sizes of the two lists, A and D, rather than the size of query result.

We also show the scalability results in Fig. 7 for query Q1, Q3, Q4, Q9 and Q10, respectively. In each subfigure in Fig. 7, the x-axis is the dataset size, and y-axis is the elapse time in msec. There are two cases. First, Fig. 7 (a), (b) and (c) show that, for query Q1, Q2 and Q3, the processing time of *HPSJ* takes the similar amount of time, despite the size of dataset increases. On the other hand, the processing time used in *IGMJ* increases linearly. It is mainly because that the result size is small. There is no time needed for *HPSJ* to check join condition whereas *IGMJ* needs more time to



(a) Q1 (b) Q3 (c) Q4 (d) Q9 (e) Q10

Fig. 7. Scalability of Reachability Query Processing over Graphs

check the join condition when the dataset becomes larger. For Q1, *HPSJ* beats *IGMJ* by nearly 100 times. Second, Fig. 7 (d) and (e) show that, for query Q9 and Q10, both the processing time of *HPSJ* and *IGMJ* increase while the size of dataset increases. It is because the result size for query Q9 and Q10 are large. *HPSJ* takes more time to retrieve data from disk with more B+-tree searches, because the number of centers can be large accordingly. Nevertheless, *HPSJ* is faster than *IGMJ* in all cases. In particular, for query Q9, almost all $item$ elements have $text$ sub-elements. It implies that *HPSJ* needs more time to retrieve data like *IGMJ* does to process the two lists. Nevertheless, *HPSJ* is about 1.4 times faster than *IGMJ*.

5.2 Reachability Query Processing over Trees

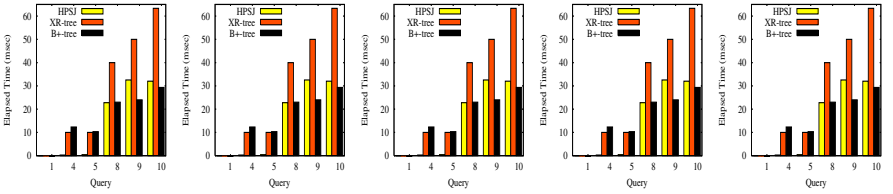
We show our testing results over XML trees in comparing our *HPSJ* algorithm with the two index-based structural join algorithms, namely, *XR-Tree* [10] and *B+-tree* [6]. Because the XML trees are used, in this testing, $A \rightsquigarrow D$ is the same as $A//D$. In this testing, we use 5 tree datasets given in Table 3. We only show our result using 6 queries, Q1, Q4, Q5, Q8, Q9 and Q10. Like Table 4, the result sizes for these queries with are listed in Table 5.

Table 5. Queries for XMark Data Sets as Trees

	$A//D$	10M		20M		30M		40M		50M	
		$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$	$\|A\|$	$\|D\|$
Q1	$africa//item$	1	55	1	110	1	165	1	220	1	275
Q4	$europe//incategory$	1	2307	1	4490	1	6696	1	8974	1	11187
Q5	$namerica//incategory$	1	3756	1	7427	1	11414	1	15261	1	19003
Q8	$item//keyword$	1510	4189	3026	8330	4512	12456	6023	16582	7483	20544
Q9	$item//text$	2175	6246	4350	12454	6525	18669	8700	24848	10875	30972
Q10	$item//incategory$	2175	8219	4350	16290	6525	24509	8700	32787	10875	40925

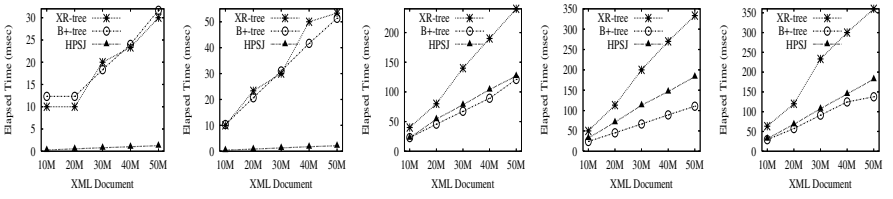
Fig. 8 shows the testing result. As can be seen in Fig. 8, *HPSJ*, *XR-tree* and *B+-tree* all perform in a similar way due to the similar I/O complexity [10]. *HPSJ* is faster than *XR-Tree* in all the cases, but is slower than the *B+-tree* algorithm in the cases where the number of results is large. The testing results for *XR-tree* and *B+-tree* show the similar trends as reported in [12]. In brief, *XR-tree* cannot perform well if they need to access the stab lists multiple times, which requires more I/O costs. *B+-tree* based algorithm needs to access most of the leaf pages of the index structure. It can be faster than *HPSJ* when the result size is large. Another reason that *B+-tree* outperforms *HPSJ* is that there are possible random I/Os when using *HPSJ*. In other words, $H^+(L_i, w)$ and $H^-(L_j, w)$ are stored in different disk pages.

Fig. 9 shows the elapsed time of processing reachability queries while increasing the size of the trees. As seen in Fig. 9 (a) and (b), when there is a small number of query results, *HPSJ* outperforms the indexed structural join algorithms at most 44.79 times and 23.99 times on average. When the query results become big, as shown in Fig. 9 (c), (d) and (e), *HPSJ* outperforms *XR-tree*, and *B+-tree* outperforms *HPSJ*, for the reasons discussed above.



(a) 10M (b) 20M (c) 30M (d) 40M (e) 50M

Fig. 8. Performance on Different Data Sets as Trees



(a) Q4 (b) Q5 (c) Q8 (d) Q9 (e) Q10

Fig. 9. Scalability on Trees

6 Conclusion

We studied reachability query processing, which is to find all elements of a type D that are reachable from some elements in another type A, denoted $A \rightsquigarrow D$. We proposed a novel approach which processes reachability queries on the fly. In other words, we do not need to check any join conditions while processing reachability queries. Our approach is based on 2-hop labeling for a directed graph G which consumes $O(|V| \cdot \log |E|)$ space. A novel join-index was proposed in this paper which is built on a small table and B+-tree. With the join-index, the high efficiency is achieved. We conducted extensive experimental studies. We showed that our approach can significantly outperform the up-to-date algorithm for reachability queries over graphs, and achieve high efficiency for processing reachability queries over trees.

Acknowledgement

The work described in this paper was supported by grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK418205).

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
2. R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of SIGMOD'89*, 1989.

3. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 141. IEEE Computer Society, 2002.
4. B. Berendt and M. Spiliopoulou. Analysis of navigation behaviour in web sites integrating multiple information systems. *The VLDB Journal*, 9(1):56–75, 2000.
5. J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. Submitted for publication, 2005.
6. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *Proc. of 28th International Conference on Very Large Data Bases (VLDB)*, pages 263–274, Hong Kong, China, August 2002.
7. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.
8. S. DeRose, E. Maler, and D. Orchard. XML linking language (XLink) version 1.0. `http://www.w3.org/TR/xlink`, 2001.
9. S. DeRose, E. Maler, and D. Orchard. XML pointer language (XPointer) version 1.0. `http://www.w3.org/TR/xptr`, 2001.
10. H. Jiang, H. Lu, W. Wang, and B. Ooi. Xr-tree: Indexing xml data for efficient structural join. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*. IEEE Computer Society, 2003.
11. I. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I. Paulsen, M. Peralta-Gil, and P. Karp. Ecocyc: A omprehensive database resource for escherichia coli. *Nucleic Acids Research*, 33(D334-D337), 2005.
12. H. Li, M. L. Lee, W. Hsu, and C. Chen. An evaluation of xml indexes for structural join. *SIGMOD Rec.*, 33(3):28–33, 2004.
13. P. Romero, J. Wagg, M. L. Green, D. Kaiser, M. Krummenacker, and P. D. Karp. Computational prediction of human metabolic pathways from the complete human genome. *Genome Biology*, 6(1):1–17, 2004.
14. R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *Proc. of EDBT'04*, 2004.
15. R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proc. of ICDE'05*, 2005.
16. A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *Proc. of VLDB'02*, 2002.
17. H. Wang, W. Wang, X. Lin, and J. Li. Labeling scheme and structural joins for graph-structured xml data. In *Proc. of The 7th Asia Pacific Web Conference*, 2005.